

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

10-17-90
506202
P. 39

**DOCUMENTING THE DECISION STRUCTURE
IN SOFTWARE DEVELOPMENT**

By

J. Christian Wild, Principal Investigator

Kurt Maly, Co-Principal Investigator

Stewart N. Shen, Co-Principal Investigator

Final Report
For the period ended August 15, 1990

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NAG-1-1026
Carrie K. Walker, Technical Monitor
ISD-Systems Architecture Branch

(NASA-CR-186865) DOCUMENTING THE DECISION
STRUCTURE IN SOFTWARE DEVELOPMENT Final
Report, period ending 15 Aug. 1990 (Old
Dominion Univ.) 39 p CSCL 093

N90-29108

Unclas
G3/61 0305303

September 1990

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

**DOCUMENTING THE DECISION STRUCTURE
IN SOFTWARE DEVELOPMENT**

By

J. Christian Wild, Principal Investigator

Kurt Maly, Co-Principal Investigator

Stewart N. Shen, Co-Principal Investigator

Final Report
For the period ended August 15, 1990

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NAG-1-1026
Carrie K. Walker, Technical Monitor
ISD-Systems Architecture Branch

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, Virginia 23508-0369

September 1990

Technical Report # TR-90-19
Decision-Based-Software Development

Chris Wild and Kurt Maly

*Old Dominion University
Department of Computer Science
Norfolk, Virginia 23529-0162
U. S. A.*

*03-16-90
(revised 9-25-90)*

Decision-Based Software Development

Chris Wild Kurt Maly
 Lianfang Liu
Department of Computer Science
Old Dominion University
Norfolk, VA 23529-0162
(804) 683-4679

September 25, 1990

Abstract

Current software development paradigms focus on the products of the development process. Much of the decision making process which produces these products is outside the scope of these paradigms. The Decision-Based Software Development (DBSD) paradigm views the design process as a series of interrelated decisions which involve the identification and articulation of problems, alternates, solutions and justifications. Decisions made by programmers and analysts are recorded in a project data base. Unresolved problems are also recorded and resources for their resolution are allocated by management according to the overall development strategy. This decision structure is linked to the products affected by the relevant decisions and provides a process oriented view of the resulting system. Software maintenance uses this decision view of the system to understand the rationale behind the decisions affecting the part of the system to be modified. The relationships between decisions help assess the impact of changing one or more decisions. We describe D-HyperCase, a prototype Decision-Based Hypermedia System and give results of applying the DBSD approach during its development.

1 Introduction

Although most people believe that the structured methods introduced in the 70's to combat the "software crisis" have increased productivity and reliability, there is evidence that such methods may contribute to the problem of developing software systems [Sne89]. A top-down design decomposition may be a good method for documenting a finished design, however it is a poor model of the design process¹, presenting an often unachievable ideal. On the other hand, most bottom-up, or compositional, approaches to software development also fail to adequately support the design process by narrowly focusing on the products of software development (such as libraries of functions or objects). Current structured engineering methods use the product structure to structure the tasks of the design process. However, many aspects of the design process have no counterpart in the product structure. The need to represent the process structure as well as the product structure has recently attracted interest [IEE88]. However, much of the emphasis in this research is on modeling the design process as a program execution in which the initial problem is transformed into a number of intermediate products culminating in the final product, the software desired. The record of the transformations performed constitutes a design plan that can be replayed when the design is reused or maintained [Bal85,Fic85]. Such a record may capture the results of a design process but does not represent how the design was actually done. Again the design process is not supported and information about the design process is lost. This loss of information makes it difficult to replay the process program [Mos86,Bal88].

In contrast to this transformational model of the design process is the problem solving model [Das89,Con88]. The problem solving model involves:

- Identification and articulation of the problem to be solved. In many cases the true problem has not been identified at the beginning of the design process. An important aspect of the design process is the discovery and analysis of the problems to be solved.
- Most problems allow for several different solutions. One of the skills of a good designer is the ability to generate alternate solutions. For some problems, only one choice among the alternates is necessary. In other cases, several choices among the alternates may be desirable (for instance, to improve performance). Because a solution to a problem may itself constitute a subproblem, the process of problem solving is recursive.
- The myriad of choices facing the designers is one reason the design process is so difficult. There is a risk that a proposed solution may in fact prove infeasible. Analyzing the alternate solutions to a problem can be very

¹We use the term design process to encompass all activities which constitute the software life cycle including requirements analysis, specification, design, implementation, validation & verification and maintenance.

time consuming. In some cases, a simulation model or prototype must be built to better understand the tradeoffs among a set of alternates. If there are several feasible solutions, choosing an optimal one may not be possible due to the large number of choices available. Often the best the designer can achieve is a solution that satisfies all the constraints. The skill and experience of the designer then determines the optimality of the proposed solutions. Much of what is termed software maintenance involves the reexamination of the set of alternates.

- The current choice of one or more alternatives.

The design process is limited by the inability of people to articulate a precise characterization of the problem they wish to solve and to thoroughly understand the consequences of their proposed solutions². This limitation means that problem requirements are frequently incomplete, inconsistent, or otherwise inappropriate expressions of the problem to be solved. Thus the design process will involve changes to the problem requirements. Software maintenance is a continuation of the design process resulting from changes in the problem requirements (adaptive), changes to correct faulty solutions to a problem (corrective) or changes to improve inappropriate solutions (perfective). A design and maintenance methodology which supports the problem solving aspects of software development must help the software engineer manage change by recording the decision rationale and the relationships between problem and solutions.

In this paper, we propose the **Decision-Based Software Development (DBSD)** paradigm to software development and maintenance. In the DBSD approach, the decision is the focal element of the problem solving process. The importance of recording the decisions made during the design process and their justifications has been recognized by others[RORL90]. The KAPTUR (Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales) system supports reuse of alternate architectures for telemetry and command control center software [CTA89]. KAPTUR records distinctive features of the recommended architecture. Distinctive features are those in which an architecture is significantly different from other architectures. The rationale for choosing the distinctive feature is also recorded. Plausibility-Driven Design was developed in order to build confidence in a proposed computer architecture before a physical realization is undertaken [AD87]. Each problem decomposition step records the knowledge used in that step and the justification for taking that step. Because performance is a critical aspect of computer architecture design, this approach integrates non-functional requirements into the design process. The automated replay of designs is an important part of the specification-based software generation paradigm [Bal85]. A recent review of work in this area identifies the need to record and utilize decisions and their justifications in order to make further

²This limitation is what Simon refers to as "bounded rationality" in synthesizing a design. [Sim81]

progress in this research [Mos89].

Section 2 presents our approach to decision based software development. In section 3, issues with respect to software maintenance are discussed. In section 4, D-HyperCase, a CASE tool to support DBSD, is described. In the following section, results on the use of our DBSD approach to the development of D-HyperCase are given. In section 6, we discuss our experiences with the DBSD method and outline several future developments we plan to undertake.

In the rest of this paper, we use the term software development to include both pre-deployment design as well as post-deployment design. The distinction between initial development and continuing maintenance is often artificial. Maintenance begins the moment that the first requirement is written down and continues throughout the life cycle. In addition, a significant part of software maintenance is development of new or corrected functions. The boundary between software reuse and adaptive maintenance is fuzzy at best. Software development by evolving existing systems may be the preferred method of developing applications in the 1990's. For these reasons, much of what will be discussed in this paper applies equally well to initial development as well as maintenance and we will use the term software development to encompass both activities.

2 A decision based paradigm for structuring the problem solving process

In the Decision Based Software Development (DBSD) paradigm, the problem solving process is documented by its decision structure (as defined in section 2.2). This decision structure provides a new way of viewing of the software document base³. Data base researchers have introduced the term 'view' to allow for different interpretations of the same data. Similarly, we propose the term 'view' to allow for different structuring of the same document. A view organizes a document, or a set of documents, from a particular perspective. For example, a **module view** describes the traditional module structure of source code. For a programming language which supports functional abstractions, the module view gives the sets of functions composing the system and their interrelationships. For other languages, the module view would be the set of data abstractions, packages or objects composing the system. While the value of the module view is well recognized in software engineering, other views of the document base are also useful. Data flow analysis views a system through the definition and use of data items. In this paper, we propose the **decision view** of the document base. This view structures the document base according to the decisions made during the problem solving process which created it. For example, the **source code decision view** is the set of source code statements

³The document base is the set of documentation supporting a software project. This includes requirements, specifications, design documents, test plans, operations manuals, etc.

pertaining to a particular decision.

2.1 The Problem Solving Process

The exploration of the problem/solution space is frequently driven by perceived risks [Boe86], special opportunities, managerial directive or personal/team preference. Issues which are not clearly understood may be explored in some depth before returning to mainstream development. In the early stages of problem solving, brainstorming can be very productive [CB88]. The order in which the problem/solution space is explored may appear arbitrary to the outside observer. Managing this highly creative process is one of the major challenges of software engineering. Some claim that one of the goals of modern software engineering is reproducibility of the products. The danger of reproducibility is that it limits creativity in the solution. Great designers should be expected to produce better solutions than less experienced and less talented colleagues. However, if the system is to be maintained by others, then the thought process which went into that great design must be captured. In this way great designers can be used to maximum advantage and others can learn from their efforts through the documented record of the design process [Bro87]. Creativity must be supported, nurtured and managed. We believe the creative aspects of the problem solving process can be structured, although not through a structure imposed by the products (that do not yet exist) but rather through support of the process which eventually will produce a structured product. Because there is a record of each step in the problem solving process, there is not the same need to record product structures early in the process.

By considering the development process to be a problem solving activity, lessons learned in the research in automated problem solving can be applied to software development. The design process is characterized by search [Das89]. This search is performed in a certain chronological order determined by the problem solving strategy. This search is characterized by dead-ends and sub-optimal solutions. Backtracking is necessary in order to recover from earlier bad choices. It is well known that backtracking in chronological order of the decisions made can be very inefficient. A bad choice made very early in the design process will require backtracking to that point with the loss of all work done after that choice. This is true even if the decisions made after the faulty one do not depend on it in any way. In order to increase the efficiency of search, one can record the dependencies between decisions and only retract those decisions which are truly dependent on the bad decision. This kind of backtracking is called **dependency directed backtracking** [Doy79]. The work on dependency directed backtracking has led to a problem solving architecture which separates the problem solver from the process which records and maintains the dependency information (commonly referred to as the **Truth Maintenance System** [dK86]). Our work adopts this architecture. The decision making is done by people while the software engineering environment maintains the doc-

ument base including the dependencies between decisions. The backtracking that occurs during the initial development as well as that which occurs during maintenance becomes more efficient and reliable because of the record of the true dependencies between decisions, problems and solutions.

2.2 The decision structure

The DBSD approach augments a traditional software development document base by a record of individual decisions and the relationship among them. An individual decision is a 4-tuple as shown in Fig. 1. These solutions in turn

Problem: description of the problem being addressed by this decision.

Alternatives: a set of possible solutions to the problem. Recording the set of alternates not chosen has several advantages:

- Promising alternates which led to dead-ends or undesirable results may prevent wasted effort during maintenance if redesign involves this decision.
- Promising alternates which were identified but not pursued due to lack of resources can provide help facilitate future improvements.

Solution: The alternate(s) chosen to solve this problem.

Justification: The reason why the particular solution was chosen. A justification can take many different forms:

- A formal proof that the solution satisfies the problem.
- An informal argument of satisfaction.
- Results from a prototype, simulation or literature review.
- Non-technical issues such as limited resources, company policy or legal restrictions may influence the decision.
- The existence of previous decisions may provide a context in which a particular choice is preferred.
- A time/space analysis.
- An estimate of engineering effort to produce a solution (cost analysis).
- Compatibility concerns with systems which must interface to this system.
- Operational compatible with fielded versions of this system.

Figure 1: Decision Object

may lead to other problems. The last decision in a chain of problem solving





should refer to some software product which documents the solution ⁴. The beginning of a well-formed problem solving chain should be some requirement, specification or need. For purposes of visualization, a decision is divided into two parts. We introduce the term **problem node** to mean a node which contains the description of a problem, and a **decision node**, a node which contains the selected solution including alternatives and justifications.

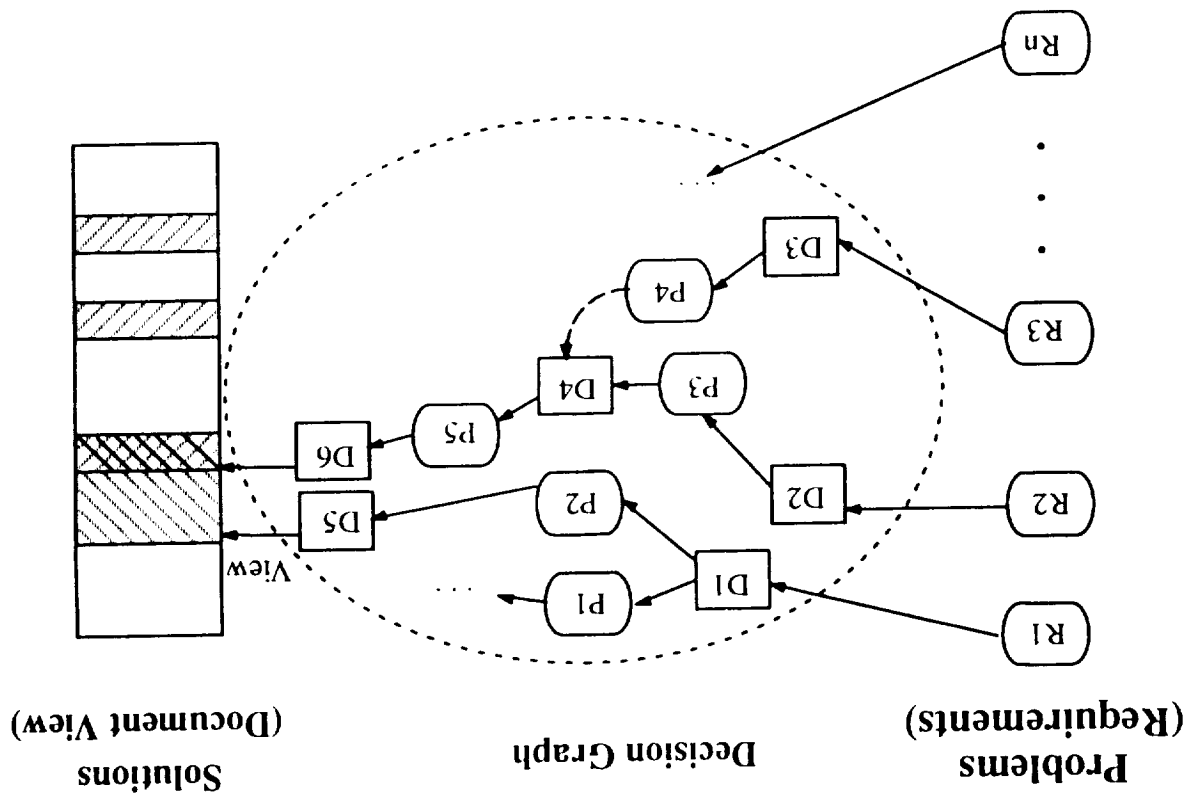
The relationship between problem and decision nodes is represented in a **decision dependency graph**. We define a **dependency link** between decisions if one decision generates a solution which becomes the problem addressed by the second decision and a **justification link** between nodes when one node is used to justify a particular decision in the second node. Justification links provide the context in which a decision is made.

The distinction between dependency and justifications is important during the change process. When a problem node is deleted, all nodes dependent only on it can also be removed because we no longer have to solve that particular problem. Typically a change in a requirement will require a change in some of the original decisions. Old solutions may no longer work and any decisions which depended on only those solutions can be discarded. Any code associated with a discarded decision is obsolete and should also be discarded. Changes in a node justifying a decision may or may not require changes in that decision. For example, the choice of a particular interprocess communications protocol, although sub-optimal, may be justified by its availability, the lack of availability of a better protocol and the limited resources available for implementing a better protocol. If at some later time, a better protocol becomes available, it is still possible to use the implementation based on the old protocol even though the justification for choosing it has gone away. Having a record of the decision and its justification helps other people understand why the system is implemented in a suboptimal manner. On the other hand, if the old protocol is removed and replaced by a new one, then the decision must be changed.

Any of the nodes in the decision dependency graph can be related to a software document by a **view link** which associates all relevant sections of the document to that node. The result is a decision graph which links requirement nodes to the relevant information in the document base as shown in Fig. 2. A requirement may be refined into one or more sub-problems. The solutions to these subproblems can be done in the context of other decisions. In the figure, requirement "R3" might represent a non-functional requirement whose refinement as "P4" justifies the functional decision "D4". The structuring provided by decisions does not necessarily correspond to the normal presentation structure of a document. The view of a decision may be scattered throughout a document impacting many parts of it (e.g. the view of decision "D6"). In addition, several decisions may impact the same part of the document. As

⁴What constitutes a solution differs with the level of abstraction of the problem solving. Specifications and design documents represent different levels of solutions. The requirement document represents a high level solution to the user's problem

 Dependency Link
 Justification Link
 Problem
 Decision



shown in the figure, the decision views of "D5" and "D6" overlap. Associating multiple views with a portion of a document helps the software engineer better understand the complex interrelationships that exist within the software system and to assess the impact of making changes to the related parts of the system.

Through the decision dependency graph, one can trace from any document back through the relevant decisions to a requirement. One can also trace from a problem to its solution, expressed perhaps as the lines of code which implement a solution to that problem. Requirements, specifications and design documents represent a solution at their respective levels of abstraction and can also be viewed through the decisions which created them. Also some of the final solutions are not programs. Users manuals and operations guides are part of the system solution and can be in the view of decisions. Fig. 3 shows the decision dependency graph divided into levels corresponding to requirements, specifications, design and implementation. This figure indicates the generality of the DBSD paradigm. All phases of the life cycle can be viewed as problem solving and the corresponding products can be viewed through its decision structure⁵. The solutions posed at one phase of the life cycle become problems to be solved at later stages of the life cycle. Although in this paper we concentrate on the decision view of the document base, other methods for viewing the set of documents may be available. The figure shows a module view and its calling structure, a data flow view, the user/system behavioral view (expressed as a context free behavior grammar), and animation figures to simulate the execution of the program.

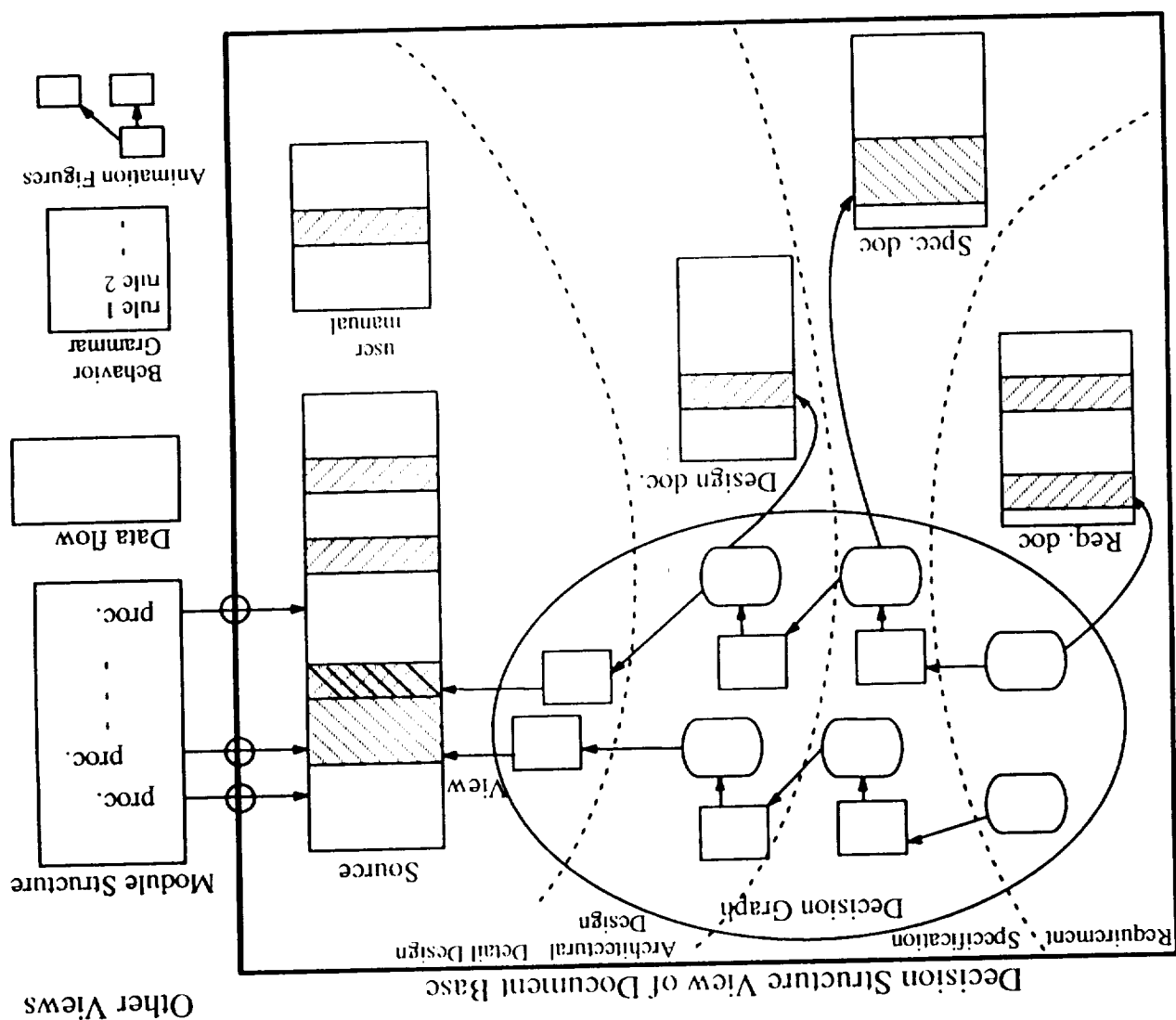
2.3 Using DBSD for Design

As much as was practical, the DBSD approach was used during the design of a prototype DBSD software engineering environment called D-HYPERCASE [WM89]. This exercise has given considerable insight into the use of the DBSD paradigm for initial development. It was anticipated that recording the decision structure would place an additional burden during the initial design, but that the increased costs would be justified by better support for maintainability. What we learned, however, was that using the DBSD paradigm could provide valuable support of the initial design process. Three advantages of the DBSD approach for the design process discussed in this section are:

- Managing the dynamics of the design process.
- Identifying incompleteness in the requirements.
- Placing a particular problem solving activity in a larger context.

⁵In the figure only one of the two views at each of the early life cycle levels is shown. This was done to reduce clutter within the figure.

Fig. 3 Decision-Based Software Documentation



One of the most important issues to be addressed in a maintenance support environment is the traceability of the program to its requirements and vice versa. This allows the software engineer to understand a program (by tracing back to requirements) or to assess the impact of changing a requirement (by tracing from requirements). Every requirement should be linked through the decision dependency graph to the source code and vice versa. Top-down and bottom-up approaches build traceability graphs which are always connected. However, it may not be possible or desirable to keep the traceability graphs connected. If the state of the problem solving process is recorded, then it is possible to schedule the problem solving process opportunistically. Thus, if an expert in security from company headquarters is visiting this week, management may choose to work on security problems even though such work may be otherwise premature. In fact, it may not be possible to choose a solution because the security requirements have not yet been sufficiently refined. However, alternate solutions could be recorded along with the justifications for choosing one over the other. Work on this problem can be recorded in a decision node with the solution slot left empty. This exploration may also indicate in what way the requirements with respect to security are vague and need to be clarified or expanded, as well as the possible interactions with other design concerns.

In some cases when solving problems, several alternates can be identified but there is not sufficient information to choose one over the other. The software engineer could choose one arbitrarily, but the danger in this is that if a similar problem must be solved somewhere else, then a different choice could lead to inconsistency (at least in style). Recognizing that one is faced with an arbitrary choice leads to the identification of a new problem in which this problem and all similar problems are specific instances relying on the same justification. For example in designing the user interface for the prototype, the problem of handling erroneous inputs was faced. There are many styles of handling error messages and it may not matter much which one is chosen in a particular program. The requirements writer may not wish to specify a particular error handling style because to do so may unnecessarily restrict the design. However, whichever style is chosen should be used consistently throughout the system (such consistency may be a requirement). Thus justifying a particular solution identifies a larger context in which this solution occurs, poses the larger problem, solves it in general and then uses that solution to justify the particular solution.

During the design of the prototype, we found a number of cases in which it was easier to develop a solution to a problem than to identify the problem itself. That is we tended to write requirements in terms of a particular solution rather than as a general problem statement (designing in the requirements). Solutions tend to be more visible than the underlying problems. By asking what problem does this requirement address, we were often able to discover when this happened. For example, the original requirements for the prototype called for one editing window and several read-only windows to display parts of the document base. The read-only windows used very different conventions than the editing

window for browsing through documents. This requirement represents a particular solution to a more general problem relating to response time. The editor which we planned to use for this project is a large program. It was anticipated that having multiple copies of the editor, one for each read-only window would be detrimental to performance. However, the program specified to manage the read only windows was an existing UNIX utility which used a very different command structure from that used by the editor. The requirements specified one way to solve the problem and in doing so raised the problem of multiple inconsistent sets of commands for the common functions to browse through a document. The real problem to be solved was how to achieve acceptable performance given that the user could have several windows open concurrently and given the non-functional requirement for a consistent user interface. By emphasizing problems, alternates, and solutions, many of these pseudo problems can be discovered.

3 How DBSD supports software maintenance

The challenge facing the maintenance engineer is indeed a formidable one. The maintenance engineer must selectively understand in sufficient detail those parts of a large system which are the focus of a particular maintenance task. The impact of alternate solutions to that task within the overall structure of the system must be assessed and the chosen solution must be implemented without violating existing constraints of the system. Given that it is impossible for anyone to completely understand any large system in intimate detail, the success of any maintenance task depends on the ability to find the relevant information in the document base. The set of relevant information needed to perform a maintenance task is referred to as the closure.

3.1 Understanding the Software Document Base

The original designer comes to an understanding of the system rather slowly and records that understanding in a series of documents. The main purpose of these documents is to help the maintenance engineer come to a similar understanding of the system. The difficulties with creating, maintaining and using documentation are well known. Many authors assume that only the source code is truly up to date and work only with the source code in maintaining a system. Also the structure of most documentation is along the lines of the products produced and it is difficult to find the relevant portions of the document base which pertains to a particular problem solving activity. This difficulty means that it is unlikely that the document base will be adequately maintained. Because of this, a major portion of the time spent on a maintenance task is spent in understanding the existing system [Cha88,WM88]. The decision structure partitions the document base into decision views. The decision closure are all

portions of the document base which are related to that decision. In the DBSD paradigm, understanding a document base means finding the relevant decisions (by browsing or keyword search) and viewing the document base through those decisions. The justifications and alternates considered, as recorded in the decision helps the maintenance engineer understand why the system is structured as it is.

Fig. 3 illustrates the relationship between the decision dependency graph and the software document base.

It is important to realize that a line of a document may be part of several views. For example, a line of code might exist because it is part of a decision on security policy and also because it calculates some function (which needs to be secure). This line would be in the view concerning security policy as well as the view concerning the function.

3.2 Assessing Change

Assessing the impact of making a change involves two considerations. First, what parts of the document base must be changed? Second, how much effort is involved in making those changes? The DBSD paradigm supports the first but leaves the assessment of effort to human judgement. Finding the relevant parts of the document base which are affected by the change is done in the understanding phase under the term closure. Unless one is adding new functionality performing a maintenance task requires that some decisions be changed. The true closure of a change is the set of statements added, deleted, or modified during the change. The closure found during the understanding phase is hopefully an upper bound on those portions of the document base which must be modified or deleted.

Although assessing effort is a matter of human judgement, the decision structure can assist in making that judgement. Work on previous alternates may have estimated effort in order to choose a solution. For example, suppose that there were two viable alternates to a problem, one which is easy to implement but relatively inefficient, and one efficient but difficult to implement, and that the simpler solution was chosen. If the maintenance task is to improve the performance in this area, then the information recorded in the decision structure may assist in assessing effort. In addition dead-end alternates which are recorded in the decision will help avoid wasted effort in maintenance.

3.3 Implementation of a change

When implementing a maintenance task one is both constrained and assisted by the existing solutions. The interrelationships between parts of a software system makes it difficult to make changes without introducing new faults. The introduction of faults into the system often occurs because of dependencies between parts of a program which appear in separate modules [LS86]. Understanding

the policy, style and functional issues which constrained the previous solution (as recorded in the justification section of the decision) will help in generating an implementation which is consistent with the rest of the system.

In implementing a maintenance task it is critically important to update all parts of the document base which are affected by the task. It is all too easy to neglect to update the supporting documentation when performing a maintenance task. Often it is difficult to find the relevant portions of the supporting documentation which require maintenance. By tying all documentation to the decision structure, we hope it will be easier to find and maintain the supporting documentation. The support environment could insist that all documentation associated with a changed decision be changed or certified as unaffected by the change. Structuring the document base through the decisions will help control the evolution of the entire document base.

Maintenance is a particularly intense form of reuse [Bas90]. Finding those portions of an existing system which can be reused is one of the challenges for a maintenance methodology. When implementing the DBSD prototype we found that the intersection of a function and a view (which we call a **frame**) is an extremely good candidate as the unit of reusing code. Several proposals have been made - procedures, templates, lines of code, abstract data type - but faults have been found with each one. In DBSD it is easy to build a new view by extracting frames from other views.

4 D-HyperCase: A Decision-Based-Support-System

D-HyperCase is a prototype software engineering environment being developed to evaluate the DBSD paradigm. This section briefly discusses D-HyperCase. A more complete description is given in [WM89].

4.1 D-HyperCase Abstract Machine

The project document base was defined in section 1 as the sum total of the recorded information about the project. The decision structure records the decisions made, alternatives considered and justifications for the solution chosen. As importantly, the decision structure provides a backbone with which to organize and access the other forms of software documentation. The ability to selectively view related parts of a set of documentation from different viewpoints is closely related to the work on hypertext [Con87]. Some of the early demonstrations of the decision based approach were done using the KMS hypermedia system [AMY88]. The two limitations to current hypertext system which were experienced were the inability to enforce a certain minimal structure on the set of documentation and the difficulty of integrating project related tools. The way we have chosen to introduce structure in the network of information comprising the project document base is to base the design on a set of typed

objects and typed links. Thus a problem object always contains a link to some decision object. The set of objects and links is user extensible. Each of the objects is managed by its own set of tools which are bound to the object at the time of object type definition. Each object must have as a minimum a display tool and an edit tool.

Since the ability to access the various objects in the project document base is a crucial part of our approach, we have chosen to build D-HyperCase on top of an Abstract HyperLink Machine (AHM) as shown in Fig. 4. This approach is similar to that taken in the design of Neptune [Big88]. The AHM maintains a data base of all the object types and their associated tools and the names of all objects according to type. Using this information, it can invoke the appropriate tool to display or edit any object (thus linking to that object). It also maintains a list of objects which the user has seen this session in order. This allows the user to retrace his steps through the document base.

The AHM supports two types of links: **structured links** and **name links**. A structured link is defined between two typed objects, one of which is the "from" object and the other is the "to" object. Whenever an object at the "from" end of the link is defined, it automatically inherits all the structured links for which its type is the "from" object. These links define the minimal structure which must exist for the new object. Thus, every object of type "problem" must have a link to a decision. On the other hand, name links provide access to objects using their name. These links are implicit and need not be represented in the AHM object base. Any object can have an arbitrary number of name links.

There is an interface to these facilities which can be accessed by any tool building on this layer. This minimal toolset provided on the Abstract HyperLink Machine layer consists of a modified Emacs text editor [Sta86] and a modified version of the Structured Graphical Knowledge Base System (SGKBS) [SLH88] under development at ODU. These tools support generic objects of the super class "text" and "graph" respectively. The minimal toolset defines one instantiation of a HyperMedia system based on AHM. D-HyperCase is then built on this layer. Further project specified tools can be built on top of D-HyperCase.

D-HyperCase is defined in terms of its objects and associated tools. These are described in section 4.2. A D-HyperCase tool can access the AHM directly through its tool interface. Such tools are called integrated tools and allow the user to directly traverse HyperLinks from within the tool (by whatever interface the tool writer wishes to provide). In addition, D-HyperCase allows the inclusion of non-integrated tools. Thus the set of tools available in D-HyperCase can be expanded to include commercially available tools. These tools would not have a direct interface into AHM. However, the user could access AHM facilities through the D-HyperCase background menu as described below. In general, the D-HyperCase tools allow display and editing of the software objects defined in the project document base in ways appropriate for that object. For example, the source code editor understands the syntax of the source language (because



ARCHITECTURAL DESIGN

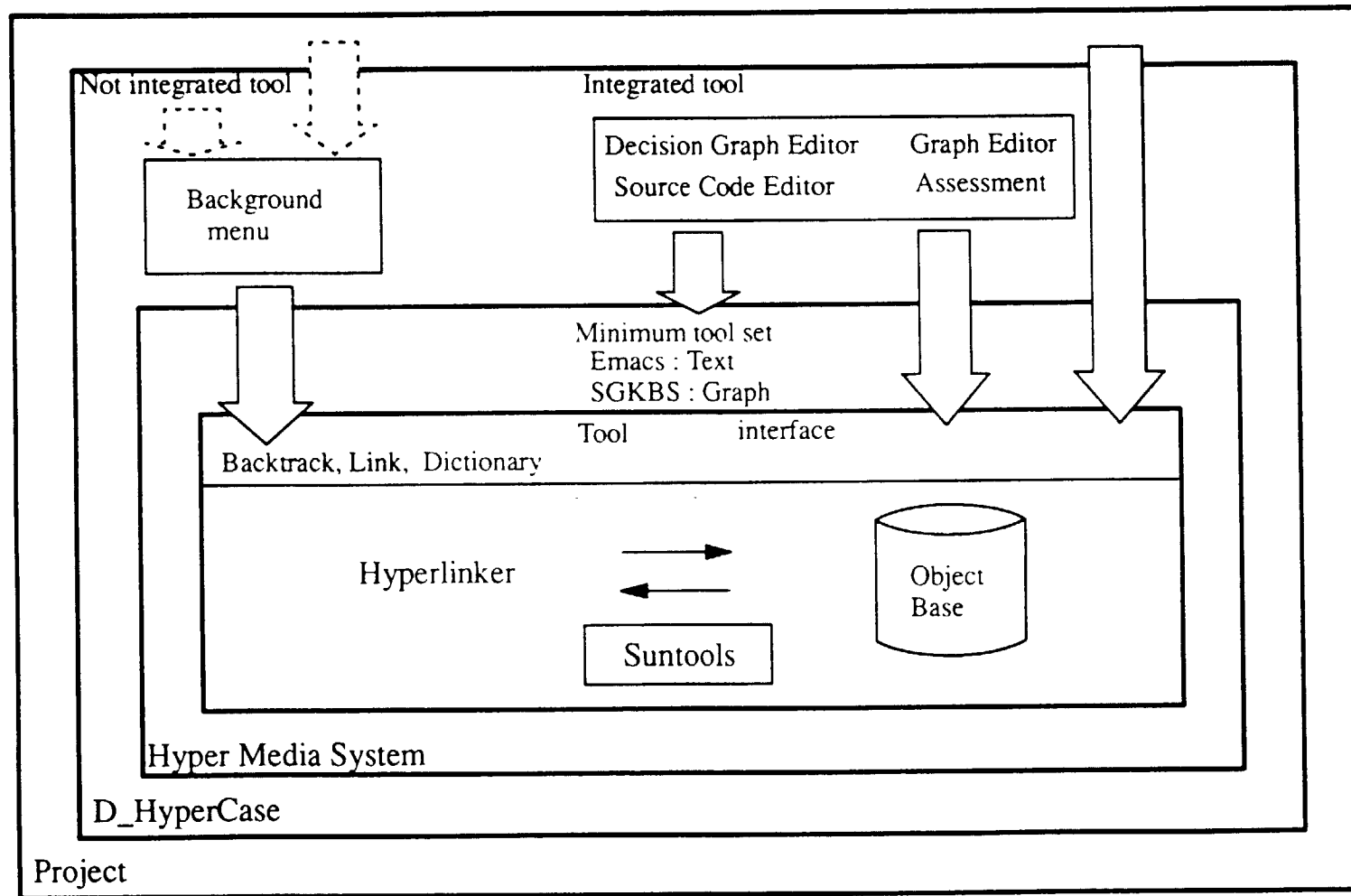


Figure 4. D_HyperCase Architectural Layers

it is a structured language editor), the relationship between source code views and decisions and allows the user to access related information through the HyperLink facility. The editor for source code objects is built upon the modified Emacs editor provided in the minimal tool set because source code is a subclass of the class of text objects.

4.2 D-HyperCase: User's Perspective

We now describe D-HyperCase from the user's perspective. Fig. 5 shows the initial screen shown to the user upon entering D-HyperCase. This screen provides an overview of the D-HyperCase System and is an object of type "document figure." Through the use of the hyperlink facility, this screen provides access to other screens which further explain the various components and usage of D-HyperCase. The top left panel shows the objects which are defined. The top right panel gives a pictorial representation of a prototypical decision graph. Further explanation is available on demand using the hypertext facility. The bottom left panel shows the set of tools which are available. The D-HyperCase tools are those provided by the basic D-HyperCase machine. The User tools are those tools which are provided on top of D-HyperCase. The bottom right panel illustrates the HyperLink connection between a decision node in the decision graph and its associated description and the set of source code views affected by this decision. Selecting a decision node in this graph will display a menu which includes the names of all HyperLinks. This allows access to the description or source code associated with this decision. A tutorial associated with this panel will lead the user through the use of the HyperLink facility in forming the closure of a decision.

Fig. 6 show the layout of the screen during a typical D-HyperCase session. The screen consists of several windows. The large underlying window contains the decision graph (which can be brought to the foreground on demand). The large window on the right is the editor (modified Emacs) window for all objects which belong to the superclass of text objects. These two windows are always open. In addition, the user may open several other windows. Fig. 6 shows several read only windows for displaying various information and a graphical editor window to be used for creating document figures. Each of these windows contain a menu for accessing both tool specific and D-HyperCase operations.

Surrounding the decision graph window is a background which can be selected to access the D-HyperCase background menu. This menu allows direct access to the hyperlink facilities, such as linking to another object or tracing back to objects previously seen. Using the background menu, the user can access the document base even when using tools which do not have a direct interface to the AHM.



D-HYPERCASE : USER PERSPECTIVE

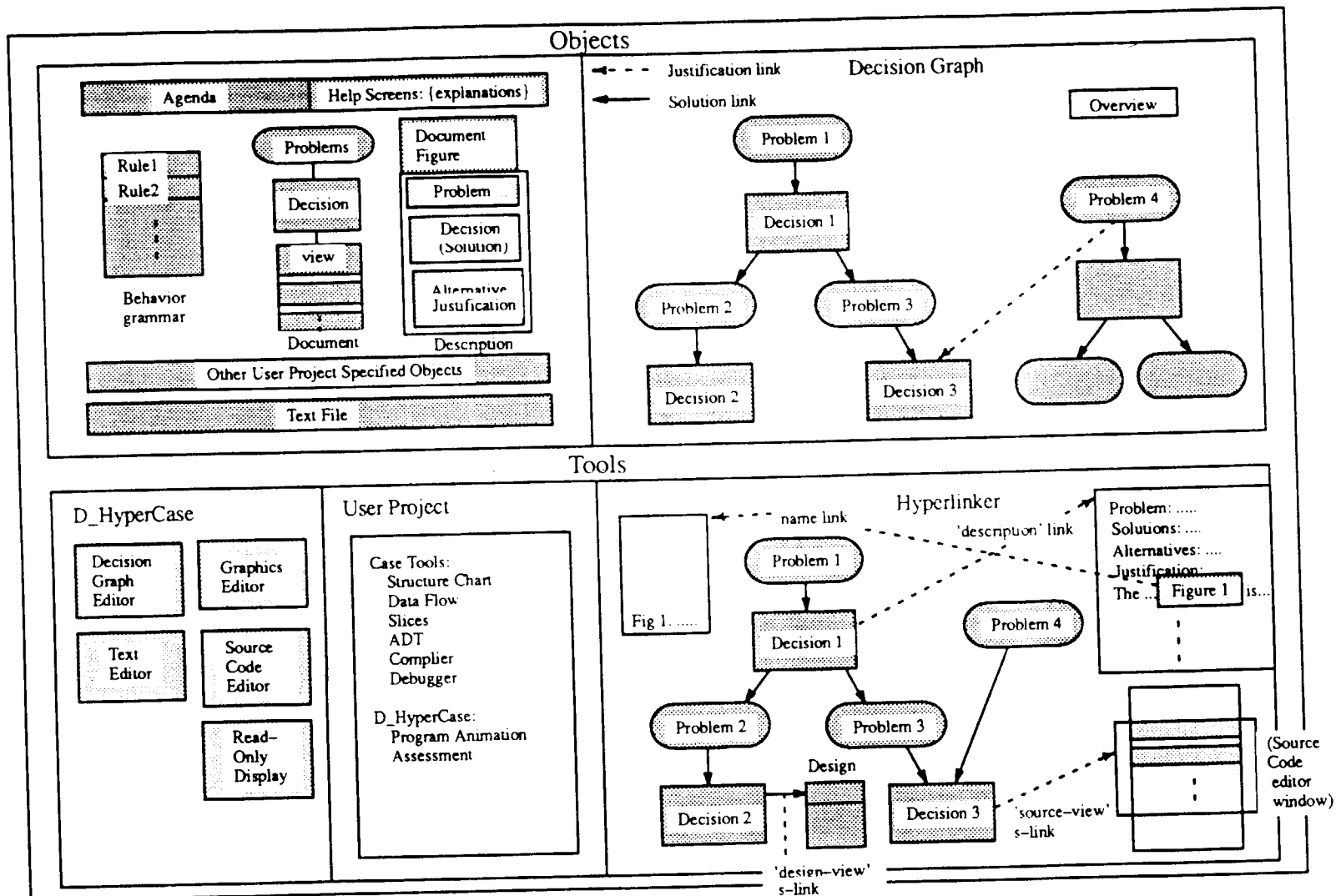


Figure 5. D_HyperCase Introductory System



D-HYPERCASE : USER PERSPECTIVE

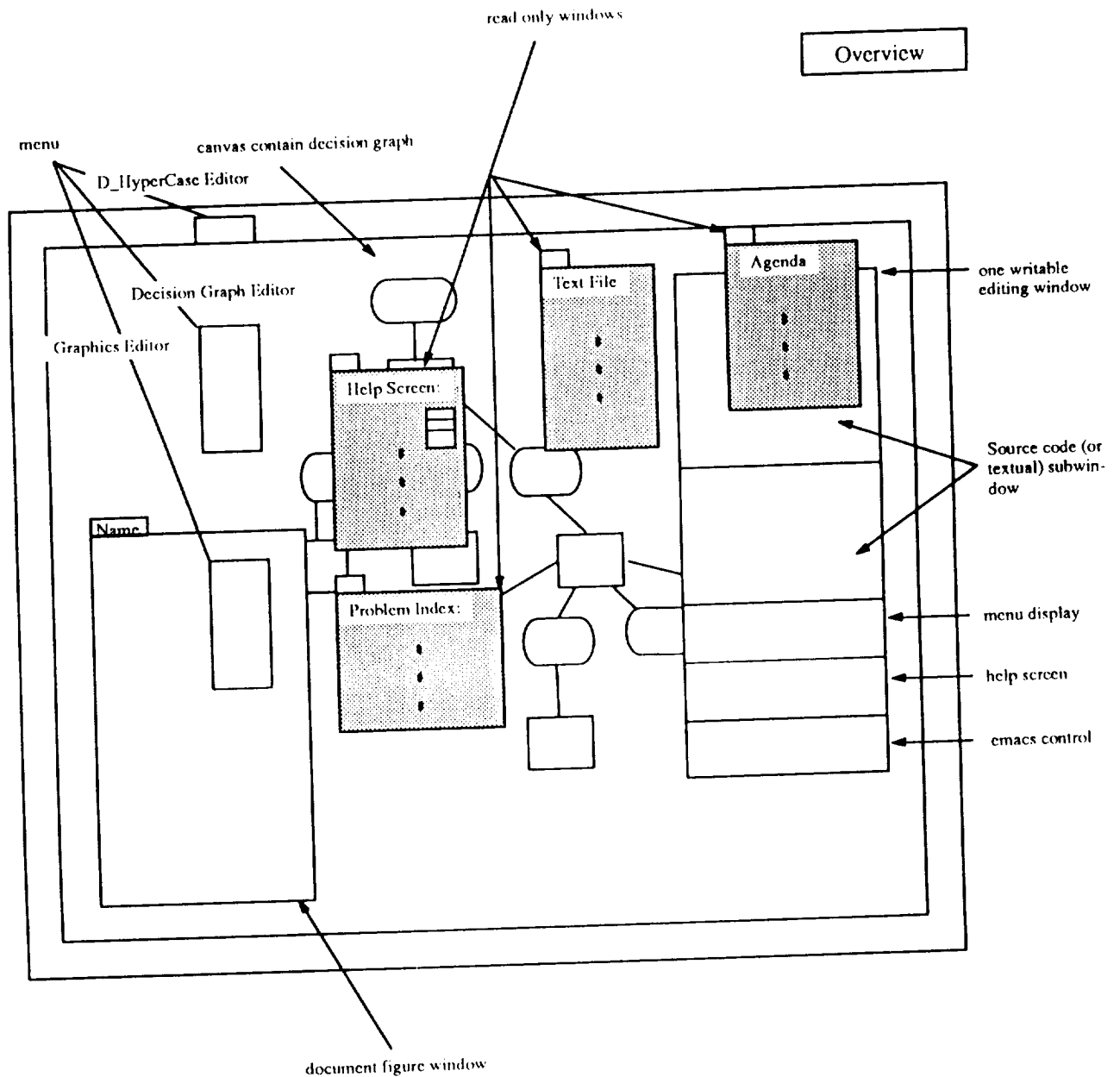


Figure 6. Layout of D_HyperCase

5 Results

We have developed D-HyperCase using the DBSD approach but since the system did not exist when the project first started, much of the documentation of decisions was done manually. Also during this time, we were still developing and refining our approach to DBSD. Despite these difficulties, we felt it was essential to gain experience with our approach and to define and refine it on a "real" system. The results reported in this section are preliminary ones based on these early efforts.

5.1 Evaluation Criteria

Performing a cost/benefit analysis in software engineering is a difficult undertaking. Many times, one must weigh present costs against future benefits. If the period of evaluation is short, then anti-regressive activities (such as reorganization and documentation) are penalized relative to progressive activities (such as adding new functionality). However, the importance of anti-regressive activities is well known. The success of current progressive activities depends on past anti-regressive activities. Lehman and Belady [BL76] have found ratio of effort between these two activities changes in recognizable cycles. spurts of progressive activities are followed by periods of anti-regressive activities to consolidate, reorganize and document for future progressive activities. Any period of evaluation should include both progressive and anti-regressive activities. We have identified four periods in which evaluation can occur.

System Life Time: A cost/benefit analysis is done over the life time of the entire system. Although this would smooth over any cyclic effects, the life cycle of many systems is too long to make this a practical period of evaluation.

Release: Most large software systems go through a set of releases over its life time. Each release usually represents a significant change in the system in which major problems are rectified and new features are added.

Task/Change Order: A task represents a unit of work to be performed. A task could be defined in response to a trouble report or could represent the addition of some new feature. Typically a task represents a work order to a single programmer or programming team and is the unit of activity allocated by management. A system release consists of many tasks.

Session: In order to perform a given task, the programmer interacts with the software development environment during a session. A session represents a contiguous period of time during which the programmer is working in the development environment. One task may take several sessions or several tasks may be completed in a single session.

One can evaluate either the products of development or the process of development or both. Product measures emphasize properties of the product independent of the method used to generate the product while process measures deal with the dynamics of the process and its effectiveness. Product evaluations involve measuring size, lines of code (LOC) for instance, reliability, and the other "ilities". Process evaluations involve such concerns as, how much effort went into understanding the problem? in implementing a solution? How many errors were introduced then corrected before the product was approved? What productivity was achieved? What level of reusability was reached? What level of effort went into unit testing vs integration testing? Which of the phases of the life cycle is least effective? most error prone? Sometimes product evaluations are used as indicators of the goodness of the process. For instance, the number of LOCs produced in a unit of time is a typical measure of productivity.

We feel that process evaluations are important in understanding the strengths and weaknesses of a method and in identifying ways in which the process can be improved. For example, some development systems are able to produce some documentation automatically and this increases productivity as measured in product output. However, if this documentation is ignored because it is difficult to understand or irrelevant, then real effort is not reduced and process productivity is unaffected. Because of the manual nature of the data collected for the results report in this section, all of the evaluations are based on product metrics. In section 6 the instrumentation of D-HyperCase to collect process measures will be discussed.

One measure of the cost of a method, is the amount of "extra" documentation that it requires. Since only the (uncommented) source code is absolutely necessary to generate a working system, all other documentation is "superfluous" to the current version. Besides the cost of the effort required to create "extra" documentation will be the cost to maintain it during the product's life cycle. We estimate that the added information is about ten percent of the source code and is roughly equivalent to the amount of comments in the source code. The number of keystrokes and mouse clicks necessary to provide the linkage between the source code and the decision is on the order of one percent of the source code. We assume that adding these linkage can be automated to a large extent by requiring that all documentation be entered in a decision view. Thus once a view is identified, all lines added are associated with that view automatically. All the updating of the links and nodes to ensure consistency is done automatically in D-HyperCase. Associating a region of an existing document to a view can be done by indicating the region in the same manner done for a cut and paste operation.

Measuring benefit is more difficult. Benefits might be measured as increased productivity, increased reliability, decreased costs or development time, increased portability or maintainability. Unless a comparative study is done against others methods, the results represent only one data point which can only be compared to general statistics reported for the software development indus-

try (such as the number of LOCs per person-month). Using product oriented metrics to measure the productivity of software maintenance tasks is distorted by the distribution of effort in the software maintenance life cycle [Cha88]. A large percent of the effort involved in a maintenance task is in understanding and analyzing the existing system [IIN90]. For instance, there is a high degree of variability in performing corrective maintenance tasks. A small oversight can be easily corrected, but a subtle problem which has plagued a mature system for several years might take quite some time to discover. Using the number of lines changed or added would not be a good measure of productivity in each case.

In section 3.1, closure was defined as those portions of the document base relevant to the performance of a maintenance task. Finding the relevant parts of the program can be a difficult task, particularly if these parts are scattered throughout the source code [LS86]. The **ideal closure** would include exactly those portions of the document base which are relevant to the task. The manner in which the document base is structured will determine what closures can be formed. The **actual closure** depends then on the structuring method and the **granularity** of view point it imposes on the document base. Documents can be structured by modules, files or objects (for source code) and chapters and sections (for other forms of documentation) or by decisions. The degree to which the actual closure matches the ideal closure can be used as a figure of merit for the structuring method. If the actual closure is a proper superset of the ideal, then more effort will be required to perform the task because the extra documentation must be understood then discarded (or worse misunderstood and changed inappropriately). On the other hand if some of the ideal closure is not contained in the actual, then it is possible that the software engineer will introduce a fault into the task because some key piece of information is missing. **Precision** refers to the degree of match between the actual and ideal closures. The measure of precision consists of two components:

1. The amount or percentage of the actual closure which is relevant to the task. If expressed as a percentage, this number represents the **density** of relevant information in the closure. One minus the density is a measure of the "noise" in the actual closure.
2. The amount or percentage of the ideal closure which is not in the actual. This is referred to as the **oversight** of the closure.

A second measure of the goodness of the actual closure is the number of structuring units which were used in forming that closure. Each structuring unit abstracts some aspect of the document base. It represents a set of related concepts in the document base as a single abstraction. If this representation helps in the understanding of the set of concepts then it is an **appropriate** abstraction. All other things being equal, the fewer abstractions one has to deal with to perform a task, the better. Carried to the extreme however, this

would lead to only one abstraction, the system itself. One also needs to measure the appropriateness of the abstraction for the task. Precision can be used to measure the appropriateness of the set of abstractions. We define the **power** of a set of abstractions relative to some maintenance task as the number of abstractions used to form the closure of that task.

In addition to counting the number of abstractions which form a closure, the **size** of an single abstraction can also be used as an indication of "abstractness". The size of an abstraction is the sum total of all documentation related to that abstraction.

5.2 Evaluation of a major release

The results reported in this section are based on an early demonstration version of D-HyperCase. This demonstration was put together by modifying and enhancing several existing software systems. A graphical editor under development at Old Dominion University was extensively modified to build and maintain the decision dependency graph. Several new functions were added to the GNU Emacs editor to edit structured text and an Abstract Hyperlink Machine was developed which allowed these two editors to communicate. For SGKBS the changes could be considered a major release of this software. Many unneeded functions were removed and new functions were added. The size of the source code went from 9000 LOC to about 6000 LOC. The decision structure for the old version of SGKBS was reengineered and for the new version was recorded. This was done before the methods and metrics of evaluation were decided to avoid unnecessary bias in creating the decision structure. The number of LOCs added, modified and deleted were counted.

This preliminary evaluation compares the functional and decision views of the system. In the functional view, it was assumed that the system was structured only through its functions and that the function was the lowest level of granularity in this structure. Thus the actual closure of a task was total LOC found in all relevant functions. In the decision view, the lowest level of granularity was the decision and the actual closure was the total LOC found in all relevant decisions⁶

Fig. 7 shows the size (in LOCs) of the average and largest abstractions in both the functional and decision view. The average function contains 57 LOCs (about one page of code). The average decision contains 220 LOCs. By the size of abstraction metric, the average decision is 3.9 times more abstract than the average function. This is a surprising result since it implies that most times the solution to a problem involves multiple functions. We had insisted that only problems which had viable alternates be recorded in the decision struc-

⁶We assume that for both the functional and decision views there exists an indexing scheme which allows the user to select the relevant functions or decisions. This is an interesting problem in its own right similar in some ways to the indexing scheme required for a reusable software library.

ture. Subproblems which were the decomposition of problems which required no decision making (that is they were purely transformational decompositions with a unique transformation) do not appear in these results. We feel that this is justified since in maintaining a system, one can only change the solution to those problems which have viable alternates.

The largest decision for SGKBS was which window system to use. The two primary choices are the window environment provided by SUN Microsystems or the emerging X window standard. Since the existing SGKBS had already chosen the SUN window environment, we stayed with this decision for the initial demonstration. However, we wanted to assess the impact of changing to X windows in the future. For an interactive graphics editing program like SGKBS, the impact of this decision is pervasive (as will be discussed in the analysis of the precision of the functional and decision views). This decision impacted almost 1700 LOC's. This is 4.7 times larger than the largest function. Again this indicates that, by the size metric, decision views are more abstract than functional views.

The activities involved in generating the new release of SGKBS resulted in the addition, modification and deletion of LOC's. Since the effort required to add, modify and delete LOC's is different, the results for the power of a set of abstractions are separated into three categories shown in Fig. 8. Again, these results indicate that the decision view is more abstract (involves fewer abstractions) than the functional view.

Although the decision view is more abstract than the functional view by the power and size measures, there is the danger it is too abstract and lacks the precision to support the maintenance process. Fig. 9 shows the precision results for the average and largest abstractions.

The average decision impacts 220 LOC's which are dispersed throughout 12 functions. Since on the average 12 functions contain 684 LOC's, the functional view requires the software engineer to read 3.1 times as much code to understand one decision as the decision view. The code for the average decision is dispersed through the related functions at a density of .32 (1 over the ratio 3.1).

The decision with the greatest impact (the choice of windowing environment) affects 1699 LOC's. These LOC's are dispersed through 105 functions (about 95% of the functions). Since these functions contain 6424 LOC's, the density of the functional closure relevant to this decision is .26 (1/3.8). Using the principle of information hiding, it could be argued that if it was known that the choice of windowing environment was subject to change, then it should have been encapsulated behind a module or package. We would disagree with the practicality of this solution in this case. Trying to hide all the window functions behind a module would be the equivalent of writing a new windowing standard (as defined by the set of interface calls to this package). This is clearly a major undertaking in its own right, probably greater than the writing of the target software. In addition, this interface must be sufficiently general to

allow alternate window environment philosophies to be hidden therein. It is not clear that this is desirable, practical or feasible. How can one anticipate all approaches to windowing? The philosophy embraced by the windowing system may result in a certain approach to solving problems. A change in philosophy would dictate that these decisions be reexamined and possibly changed to best take advantage of the new philosophical approach. Hiding the features which one environment supports well will result in inefficiencies. In fact, one of the reasons for changing from one window system to another is to take advantage of its unique features. Where feasible one should hide decisions, but this is not always feasible or desirable.⁷

Fig. 10 gives the precision results for LOCs modified and deleted from both the decision and functional views.

In changing SGKBS, 233 LOCs were modified, 2936 LOCs were deleted and 536 LOCs were added. The added LOCs were in the new decisions and new functions identified in Fig. 8. The modified LOCs were contained in 100 functions and 8 decisions containing 7860 LOCs and 5350 LOCs respectively. Although the relevance density of the decision view is better than the functional view, neither view was particularly good at precisely identifying the LOCs which must be modified. Although only 233 LOCs were in the implementation closure with respect to modification, we had no way of measuring the understanding and assessment closures. Clearly the software engineer will examine and analyze LOCs which will not be changed or deleted and that this is necessary in understanding the program and assessing impact. However, we did examine some of the decisions changed in more detail to determine why the decision closure seemed so imprecise. We discovered that in the modified SGKBS new problems were being introduced which had not been identified in the old system. For example in the old system the shapes of the graphical objects had no meaning. In the menu used to create objects, the names displayed there referred to the geometric shape (such as rectangle or oval). In the modified SGKBS, we use these shapes for special purposes, thus an oval represents a problem node and a rectangle represents a decision node. The new system introduces the new problem of associating shapes with certain semantic objects. This problem is largely unrelated to the problems of displaying and manipulating geometric objects. Since in the old system this problem had not been anticipated, the code associated with generating menu labels was part of a much larger decision on the geometrical shapes of objects and their display (which geometric objects should and could be displayed). Once the new problem of associating a semantic object (problem or decision node) with a geometric shape was introduced, the precision of that decision increased dramatically.

Since it is unlikely that all uses of the system will be anticipated in the original design, we envision a growth in the precision of the problem solving as

⁷One solution to the windowing change in SGKBS is to emulate the sunview windowing system in X. In fact SUN distributes such an emulator but because of incompatibilities some changes in the program must still be undertaken.

a result of maintaining the system. The ability to refine the decision structure through use is important for initial design and for the reverse engineering of existing designs. It would be a mistake to burden the initial design team with the task of identifying and documenting all potential decision points in the system. Rather, they should document only those decisions in which various alternates were actively investigated and analyzed. If during maintenance viable alternates to certain design problems become evident or new problems become identified, then the decision structure can be refined to reflect this new insight. Thus one may discover a useful decision substructure on a problem which originally appeared to have been a monolithic solution. With respect to reverse engineering the decision structure on an existing software system, the possibility for rapid growth of precision will mean that a useful decision structure can be identified on those portions of the system which are subject to the most change.

The original version of SGKBS contained a number of features which were not needed for D-HyperCase. Removing these features was a major part of the work done in converting SGKBS. The unneeded features corresponded to 18 problems which no longer needed to be solved. Removing these decisions and the related code deleted almost 3000 LOC's. From the functional view, 75 functions (containing 2119 LOC's) could be eliminated entirely. The row marked "functional(deleted)₁" gives results for the closure consisting only of those functions which could be deleted in their entirety. As can be seen, this closure overlooked a large proportion of the obsolete code. In the row marked "functional(deleted)₂" all functions containing obsolete code are included in the closure. In this case the relevance density is low. This again indicates that the functional view does not map well onto the activities which are performed during maintenance. For deleted code, the decision view is significantly more precise than the functional view.

6 Discussion

We believe that the DBSD approach to the development of software systems represents a significant and novel alternative to existing software development paradigms. However our experience with this method is still limited. There is still much work to be done in developing guidelines for using a DBSD approach and in integrating DBSD into a software development methodology. D-HyperCase was built to serve as an experimental medium for exploring and developing the DBSD paradigm. However, in any large system one runs the risk of burying good ideas among bad ones or in a poor implementations. Simple evaluations, such as reported in the previous section, will not uncover subtle problems or advantages. In this regard, anecdotal evaluation is often important in focusing attention on the critical issues. In this section we discuss our experiences in developing D-HyperCase and our plans to instrument it to collect more detailed data for further evaluation.

The DBSD approach is not a panacea for all the ills plaguing the software engineering community. While we claim the decision structure can be used to provide multiple "natural" views of the software system relevant to the development process, the definition of the decision graph is neither easy nor obvious at first try. But the generation of complex systems is not an easy undertaking [Bro87,Par85]. The determination and articulation of the decision structure requires an experienced person, one who can undercover the underlying problems and relationships which justify certain solutions. There is a tendency to structure the decision graph using the temporal order in which decisions were made instead by necessary dependencies. Also there is the tendency to identify decisions with functions. There is also no reason to believe that the decision graph is unique. A more experienced designer will explore more alternatives, see more relationships between problems and identify better justifications than a less experienced colleague. Documenting the decision structure places an extra burden on the software engineer. Documentation is the price one pays for increased maintainability. Our experience indicates that documenting the decision structure does support the design process by focusing on the identification of alternates and justifications.

Reverse engineering the decision structure of an existing system is difficult and in some cases impossible. The decision structure is not in the products of development. Only partial evidence of the decision making may be present. In many cases, one will have to guess at the rationales behind certain decisions. It is unlikely that reverse engineering could be automated for the foreseeable future. The difficulty in reverse engineering the decision structure is one of the reasons software maintenance is so difficult. The DBSD approach does not necessarily make the reverse engineering easier, but it does allow the software engineer to record the insights that are gained during a maintenance task with respect to the decision structure so that the next time maintenance is done on the same problems, this insight is available.

6.1 DBSD Methodology

We believe that the DBSD approach to software development can be incorporated in many different methodologies. In this section, several observations about the influence of the DBSD paradigm on methodology will be made. Many of these observations relate to the manner in which the DBSD approach allows concurrent progress on different phases of the life cycle.

Solution First. It is often easier to identify the solution than the problem it solves. This difficulty is not particular to the decision based approach. It takes great skill to uncover the real problem when given a list of wants and desires. End users find it hard to articulate their real needs but can often recognize when a proposed solution meets or doesn't meet those needs.

Generalization of Problem. In many cases, the solution to a particular problem can be satisfied by generalizing the problem, solving the general prob-

lem then instantiating the general solution for the original problem. When we first started development of D-HyperCase, we had only two tools, Emacs and SGKBS and no separate HyperLink machine. We first tried to build hyperlinks between Emacs and SGKBS before we realized that we could solve the more general problem, resulting in the HyperLink Machine, and use it to solve our particular problem. Once the more general problem is solved, extending D-HyperCase to other tools is much simpler since communication from a tool is always to the Abstract HyperLink Machine and doesn't change as more object types, operations and tools are added.

Finding a Rationale. Sometimes the alternates to solving a problem can be identified and analyzed but there is no obvious preferred choice. In such cases, one could choose one of the alternates arbitrarily. The lack of clear criteria for choosing an alternate points to an incompleteness in the requirements. In developing D-HyperCase we found that developing a rationale often involved the refinement of a general non-functional requirement.

Non-functional Problem Solving. Rationalizations tend to involve non-functional requirements such as performance or user friendliness. Addressing non-functional requirements involves a different style of problem solving from that which addresses functional requirements. For functional requirements there is usually a criteria for determining whether or not the solution satisfies the criteria. Also only one alternate is chosen as a solution even though there may be several viable ones (in the problem solving graph, the alternates for a functional problem form an **exclusive-or** node). For non functional requirements, the criteria for choosing among alternates is less absolute. If the performance requirement is for a response time of 5 seconds, an actual response time of 5.1 seconds may be acceptable. The criteria for deciding that a program meets the requirement for user friendliness is even less precise. For some non-functional requirements, several alternates may be chosen simultaneously. Achieving a performance requirements may require the development of several algorithms and data structures. Achieving user friendliness is also a matter of degree. In the problem solving graph, the alternates for non functional problems may form an **inclusive-or** node.

Articulating the Requirements. The generalization of problems, the development of rationale and the refinement of non-functional requirements indicate a style of problem solving which goes from particular instances to general problems and solutions. Often in this process it is discovered that the original problem requirements were incomplete. We consider the identification and articulation of true problems to be an important aspect of software development. Unless a project involves the development of software which is well understood, development methodologies which assume that the requirements can be accurately stated beforehand are bound to fail [Sne89].

Opportunistic Scheduling. We have not found the design process for D-HyperCase to be top down and we suspect that most design is not. During the creative ferment of design, people work at many levels, exploring critical issues

in depth until a solution path becomes clear. In the process of design, new problems may be identified whose solution one may want to defer. Sometimes the alternates are identified and analyzed but the final solution is deferred. If missing requirements are discovered, then the solution of a particular problem may have to be deferred until the requirements are formally modified. We have found that work is done at many levels of the design concurrently and that there can be many unresolved issues present simultaneously. We have found that the decision based approach helps to organize and structure this process. As problems are identified, alternatives rejected and decisions made, they are recorded. It is not necessary to immediately associate these (partial) decisions in the decision dependency graph. Instead, these decisions can be placed into a project agenda for later resolution. It is the responsibility of management to allocate resources to the decisions on the project agenda which will lead to their eventual solution and proper placement in the decision structure.

Avoiding Bias. Since every solution should be connected to the problem it solves, there should be a clear path from source code to requirements. Solutions which cannot be traced back to some problem in the requirements indicate either a missing requirement, a non problem or an implementor added feature. In the first case the requirements should be changed. In the second case the solution to the non-problem should be eliminated. In the last case, the user may choose to keep this added feature but not change the requirements. This feature is then justified as a useful but non essential part of the program. In subsequent maintenance, these features could be removed if it is found that their costs have become unacceptably high.

Support for Continued Development. We have found recording alternatives to be valuable both to document dead ends and to provide starting points for later improved designs. This later is especially useful if one is using an incremental build philosophy.

6.2 Granularity of the Decision Structure

One of the problems in using the DBSD approach to software development is its generality. Almost every action can be thought of as making a decision to solve some problem. Many of these decisions are trivial or unimportant and documenting them would require more effort than any potential benefit could justify. Examples of such decisions are what names to use for temporary variables or how to split a complex expression across multiple lines. Such trivial decisions do not need to be recorded either because the decision making process can be easily reconstructed or there is little impact in changing them. Recording trivial decisions not only places an unnecessary burden on the designer but also serves to overwhelm the maintainer with trivial documentation. Developing useful guidelines for the recording of decisions is one area which needs further research.

In other cases a problem will decompose into subproblems for which there

are no viable alternates. For example, if a stack is chosen as the solution to some data structuring problem, then designing the stack decomposes into designing the initiation, push and pop operations. There is no choice about having these operations⁸. A decomposition for which there are no viable alternates is called a **design decomposition** and a decomposition for which there are viable alternates requiring a decision and justification is called a **decision decomposition**. Our first attempts to build a decision dependency graph for D-HyperCase included both design and decision decompositions. We found such graphs to be unwieldy and repetitive of information contained in other documentation. We now require that only decision decompositions be included in the graph. A design document is still produced in conjunction with the decision graph and we introduce a 'design' link between an element in the design document and a node in the decision graph.

6.3 Evaluation of D-HyperCase

It is of utmost importance that the cost to the user of any new method or tool be sufficiently low to justify the advantages claimed. We have claimed five major advantages: ease of understanding, a handle at assessing the impact of a proposed change, traceability of source code through documentation to requirements, ease in reusing existing code for modification and support for creative design work. These are major benefits but only if a system exists to help gather and maintain the necessary information as well as allow for consistency checks.

The statistics estimated in section 5 reflect a static evaluation of the decision structure. In order to understand the dynamics of the design process, we are instrumenting D-HyperCase to collect statistics on effort and benefit. While the closure of a task is a static concept, we believe there may also be a dynamic concept akin to the working set model of virtual memory. The **working set** of a closure is that subset of the documentation which the user needs to access simultaneously in order to perform some part of a task. The ability to simultaneously view information in several windows is one way a software engineering environment might support a working set model. As of now, we don't know if the working set model is appropriate, or, if it is, how big a working set should be. We hope to gain some insight into these issues through further experimentation with D-HyperCase.

⁸There are choices about the underlying data structures chosen to implement a stack. These choices affect all the operations on stacks. Again we see that decisions do not necessarily map into functions or operations.

7 Conclusions

The Decision Based Software Development Paradigm is being developed to support the process of developing and maintaining software systems. In this paradigm, the design process is considered a problem solving activity in which problems are identified, alternates proposed and analyzed and a decision of a particular solution is made. By recording this process, both initial development and subsequent maintenance is facilitated. Our experiences in applying this paradigm to the development of a prototype software engineering environment indicate that it supports a design process which allows progress in many different phases of the software life cycle simultaneously. In particular, this approach aids in the development and articulation of the systems requirements. This paradigm makes such concurrent activity possible by making the process more visible and therefore more manageable. Non-functional problem solving plays an important role in the design process.

Preliminary results on the use of the DBSD paradigm to support the maintenance process indicate that the decision structure provides an alternate and more abstract view of the software document base. In addition the precision of a decision view for finding the documentation relevant to a particular maintenance task is greater than that of a functional view. This precision should aid in the maintenance of ancillary documentation since the decision view includes all relevant documents, providing a linkage from requirements to source code.

In order to further develop a methodology based on the DBSD paradigm, an evaluation of its use in the design and maintenance process is being undertaken. Criteria for evaluating the dynamics of the development process have been proposed and we are instrumenting D-HyperCase, the prototype DBSD environment, to collect evaluation data.

8 Acknowledgements

This research was supported by NAVMASSO and NASA Langley Research Center through contract NAS1-18584-11 and NASA Langley Research Grant NAG-1-1026.

References

- [AD87] Ulises Agnero and Subrata Dasgupta. A plausibility-driven approach to computer architecture design. *CACM*, 30(11):922-932, November 1987.
- [AMY88] Robert Akseyn, Donald McCracken, and Alise Yoder. Kms: a distributed hypermedia system for managing knowledge in organizations. *CACM*, 31(7):820-835, July 1988.

- [Bal85] R. Balzer. A 15 year perspective on automatic programming. *IEEE Trans. on Soft. Eng.*, SE-11(11):1257-1267, Nov. 1985.
- [Bal88] Robert Balzer. Processing programming: passing into a new phase. *Proc. of the 4th Int. Software Process Workshop*, 43-45, May 1988.
- [Bas90] Victor Basili. Viewing maintenance as reuse oriented software development. *IEEE Software*, 7(1):19-25, January 1990.
- [Big88] James Bigelow. Hypertext and case. *IEEE Software*, 23-27, March 1988.
- [BL76] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, (3):225-252, 1976.
- [Boe86] Barry Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14-24, Aug. 1986.
- [Bro87] Fred Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20:10-20, April 1987.
- [CB88] Jeff Conklin and Michael Begeman. *qIBIS: A Hypertext Tool for Exploratory Policy Discussion*. Technical Report, MCC, March 1988.
- [Cha88] Ned Chapin. Software maintenance life cycle. *Proceedings of the Software Maintenance Conference*, 6-13, October 1988.
- [Con87] Jeff Conklin. *A Survey of Hypertext*. Technical Report Rev. 2, MCC, Dec. 1987.
- [Con88] Jeff Conklin. *Design Rationality and Maintainability*. Technical Report, MCC, June 1988.
- [CTA89] Incorporated CTA. *The KAPTURE Environment: An Operations Concept*. Technical Report NAS5-30680, NASA Goddard Space Flight Center, August 1989.
- [Das89] Subrata Dasgupta. The structure of design processes. In M.C. Yovits, editor, *Advances in Computers*, pages 1-68, Academic Press, New York, 1989.
- [dK86] Johan de Kleer. Problem solving with the atlas. *Artificial Intelligence*, 28, 1986.
- [Doy79] Jon Doyle. Truth maintenance systems. *Artificial Intelligence*, 12(3):231-272, 1979.

- [Fic85] Stephen F. Fickas. Automating the transformational development of software. *IEEE Trans. on Software Engineering*, SE-11(11):1268-1277, Nov. 1985.
- [HN90] Mehdi Harandi and Jim Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74-81, January 1990.
- [IEE88] IEEE. Representing and enacting the software process. *Proceedings of the 4th International Software Process Workshop*, May 1988.
- [LS86] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41-49, May 1986.
- [Mos86] Jack Mostow. Why are design derivations hard to replay? In Jaime Carbonell, Tom Mitchell and Ryszard Michalski, editors, *Machine Learning: A Guide to Current Research*, pages 213-218, 1986.
- [Mos89] Jack Mostow. Design by derivational analogy: issues in the automated replay of design plans. *Artificial Intelligence*, 40:119-184, 1989.
- [Par85] David Lorge Parnas. Software aspects of strategic defense systems. *American Scientist*, 73:432-440, Sept-Oct. 1985.
- [RORL90] Spencer Rugaber, Stephen Ornburn, and Jr. Richard LeBlanc. Recognizing design decisions in programs. *IEEE Software*, 7(1):46-54, January 1990.
- [Sim81] Herb Simon. *Science of the Artificial*. MIT Press, Cambridge, Mass., 1981.
- [SLH88] S. N. T. Shen, L. Liu, and J. Hsu. A hyper-graphics tool for multidisciplinary applications. *Proc. of the Int. Symp. Expert Systems Theory and Their Applications*, Dec. 1988.
- [Sue89] Harry Sued. The myth of "top down" software development and its consequences for software maintenance. *Proc. Conf. on Software Maintenance*, 22-31, Oct. 1989.
- [Sta86] Richard Stallman. Gnu emacs manual. October, 1986.
- [WM88] Christian Wild and Kurt Maly. Towards a software maintenance support environment. *Proceedings of the Software Maintenance Conference*, 80-85, October 1988.
- [WM89] Christian Wild and Kurt Maly. Decision-based software development: design and maintenance. *Proceedings of the Conference on Software Maintenance*, 297-306, October 1989.

Case	Function (in LOC)	Decision View (in LOC)	Ratio
Average	57	220	3.9
Largest	360	1699	4.7

Figure 7: Size of Abstractions

Type	# Functions	# Decisions	Ratio
Modified	100	8	6.2
Deleted	75	18	4.2
Added	7	4	1.8

Figure 8: Power of Closure Set

Class	Functional LOC	Decision View LOC	Ratio	# F
Average	684	220	3.1	12
Largest	6424	1699	3.8	105

Figure 9: Precision of Average and Largest Abstractions

View Point (Class)	# Abstractions	Size Closure (LOC)	Relevance (Percent)	Oversight (Percent)
Functional(Modified)	100	7860	2.96	0
Decision (Modified)	8	5350	4.35	0
Functional(Deleted) ₁	75	2149	100	26.8
Functional(Deleted) ₂	102	7206	40.74	0
Decision(Deleted)	18	2936	100	0

Figure 10: Precision for Modified and Deleted LOC's
